Automata-Based Verification of Programs with Tree Updates*

Peter Habermehl[1], Radu Iosif[2], and Tomas Vojnar[3]

[1] LIAFA/Université Paris 7, 175 rue du Chevaleret, 75013 Paris, France
haberm@liafa.jussieu.fr
[2] VERIMAG/CNRS, 2 Avenue de Vignate, 38610 Gières, France
iosif@imag.fr
[3] Brno University of Technology, Bozetechova 2, CZ-612 66 Brno, Czech Republic
vojnar@fit.vutbr.cz

Abstract. This paper describes an effective verification procedure for imperative programs that handle (balanced) tree-like data structures. Since the verification problem considered is undecidable, we appeal to a classical semi-algorithmic approach in which the user has to provide manually the loop invariants in order to check the validity of Hoare triples of the form $\{P\}C\{Q\}$, where P, Q are the sets of states corresponding to the pre- and post-conditions, and C is the program to be verified. We specify the sets of states (representing tree-like memory configurations) using a special class of tree automata named Tree Automata with Size Constraints (TASC). The main advantage of using TASC in program specifications is that they recognize non-regular sets of tree languages such as the *AVL trees*, the *red-black trees*, and in general, specifications involving arithmetic reasoning about the lengths (depths) of various (possibly all) paths in the tree. The class of TASC is closed under the operations of union, intersection and complement, and moreover, the emptiness problem is decidable, which makes it a practical verification tool. We validate our approach considering red-black trees and the insertion procedure, for which we verify that the output of the insertion algorithm is a *balanced* red-black tree, i.e. the longest path is at most twice as long as the shortest path.

1 Introduction

Verification of programs using dynamic memory primitives, such as allocation, deallocation, and pointer manipulations, is crucial for a feasible method of software verification. In this paper, we address the problem of proving correctness of programs that manipulate balanced tree-like data structures. Such structures are very often applied to implement in an efficient way lookup tables, associative arrays, sets, or similar higher-level structures, especially when they are used in critical applications like real-time systems, kernels of operating systems, etc. Therefore, there arised a number of such search tree structures like the AVL trees, red-black trees, splay trees, and so on [7].

Tree automata [6] are a powerful formalism for specifying sets of trees and reasoning about them. However, one obstacle preventing them from being used currently in

* This work was supported in part by the French Ministry of Research (ACI project Securité Informatique) and the Czech Grant Agency (projects GA CR 102/04/0780 and 102/03/D211).

H. Hermanns and J. Palsberg (Eds.): TACAS 2006, LNCS 3920, pp. 350–364, 2006.

program verification is that imperative programs perform destructive updates on selector fields, by temporarily violating the fact that the shape of the dynamic memory is a tree. Another impediment is the fact that tree automata represent regular sets of trees, which is not the case when one needs to reason in terms of *balanced* trees, as in the case of AVL and red-black tree algorithms.

In order to overcome the first problem, we observe that most algorithms [7] use *tree rotations* (plus the low-level addition/removal of a node to/from a tree) as the only operations that effectively change the structure of the input tree. Such updates are usually implemented as short low-level pointer manipulations [16], which are assumed to be correct in this paper. However, their correctness can be checked separately in a different formalism, such as [17], or by using tree automata extended with additional "routing" expressions on the tree backbone as in [11].

The second inconvenience has been solved in the present paper by introducing a novel class of tree automata, called Tree Automata with Size Constraints (TASC). TASC are tree automata whose actions are triggered by arithmetic constraints involving the *sizes* of the subtrees at the current node. The size of a tree is a numerical function defined inductively on the structure, as for instance the height, or the maximum number of black nodes on all paths, etc. The main advantage of using TASC in program specifications is that they recognize non-regular sets of tree languages, such as the *AVL trees*, the *red-black trees*, and in general, specifications involving arithmetic reasoning about the lengths (depths) of various (possibly all) paths in the tree. We show that the class of TASC is closed under the operations of union, intersection and complement. Also, the emptiness problem is decidable, and the semantics of the programs performing tree updates (node recoloring, rotations, nodes appending/removal) can be effectively represented as changes on the structure of the automata.

Our approach consists in writing pre- and post-condition specifications of a (sequential) imperative program and asking the user to provide loop invariants. The verification problem consists in checking the validity of the invariants and of the Hoare triples of the form $\{P\}C\{Q\}$ where P,Q are the sets of configurations corresponding to the pre- and post-condition, and C is the program to be verified. We need to stress the fact that here P and Q are languages accepted by TASC, instead of logical formulae, as it is usually the case with Hoare logic. The validity of the triple is established by computing the set of states reachable from a state in P by executing C, i.e. $post(P,C)$, and then deciding whether $post(P,C) \subseteq Q$ holds.

We have validated our approach on an example of the insertion algorithm for the red-black trees, for which we verify that for a balanced red-black tree input, the output of the insertion algorithm is also a balanced red-black tree, i.e. the number of black nodes is the same on each path.

Related Work. Verification of programs that handle tree-like structures has attracted researchers with various backgrounds, such as static analysis [12], [16], proof theory [4], and formal language theory [11]. The approach that is the closest to ours is probably the one of PALE (Pointer Assertion Logic Engine) [11], which consists in translating the verification problem into the logic SkS [15] and using tree automata to solve it. Our approach is similar in that we also specify the pre-, post-conditions and the loop invariants, reducing the validity problem for Hoare triples to the language emptiness

problem. However, the use of the novel class of tree automata with arithmetic guards allows us to encode quantitative properties such as tree balancing that are not tackled in PALE. The verification of red-black trees (with balancing) is reported also in [2] by using hyper-graph rewriting systems. Two different approaches, namely net unfoldings, and graph types, are used to check that red nodes have black children and that the tree is balanced, respectively.

The definition of TASC is the result of searching for a class of counter tree automata that combines nice closure properties (union, intersection, complementation) with decidability of the emptiness problem. Existing work on extending tree automata with counters [8, 18] concentrates mostly on *in-breadth* counting of nodes with applications on verifying consistency of XML documents. Our work gives the possibility of *in-depth* counting in order to express balancing of recursive tree structures. It is worth noticing that similar computation models, such as alternating multi-tape and counter automata, have undecidable emptiness problems in the presence of two or more 1-letter input tapes, or, equivalently, non-increasing counters [13]. This result improves on early work on alternating multi-tape automata recognizing 1-letter languages [9]. However, restricting the number of counters is problematic for obtaining closure of automata under intersection. The solution is to let the actions of the counters depend exclusively on the input tree alphabet, in other words, encode them directly in the input, as size functions. This solution can be seen as a generalization of Visibly Pushdown Languages [1] to trees, for singleton stack alphabets. The general case, with more than one stack symbol, is a subject of future work.

1.1 Running Example

In this section, we introduce our verification methodology for programs using balanced trees. Several data structures based on balanced trees are commonly used, e.g. AVL trees. Here, we will use as a running example red-black trees, which are binary search trees whose nodes are colored by red or black. They are approximately balanced by constraining the way nodes can be colored. The constraints insure that no maximal path can be more than twice as long as any other path. Formally, a node contains an element of an ordered data domain, a color, a left and right pointer and a pointer to its parent. A *red-black tree* is a binary search tree that satisfies the following properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf is black.
4. If a node is red, both its children are black.
5. Each path from the root to a leaf contains the same number of black nodes.

An example of a red-black tree is given in Figure 1 (a). Because of the last condition, it is obvious that the set of red-black trees is not regular, i.e. not recognisable by standard tree automata [6]. The main operations on balanced trees are searching, insertion, and deletion. When implementing the last two operations, one has to make sure that the trees remain balanced. This is usually done using tree rotations (Figure 1 (b)) which can change the number of black nodes on a given path. The pseudo-code of the inserting operation is the following (see [7]):

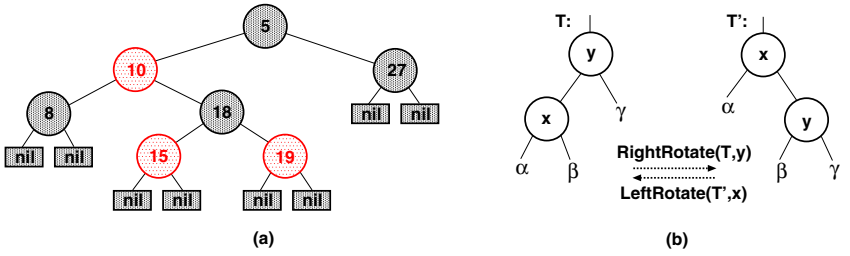

Fig. 1. (a) A red-black tree—nodes 10, 15, and 19 are red, (b) the left and right tree rotation

```
RB-Insert(T,x):
  Tree-Insert(T,x);   % Inserts a new leaf node x
  x->color = red;
  while (x != root && x->parent->color == red) {
    if (x->parent == x->parent->parent->left) {
      if (x->parent->parent->right->color == red) {
        x->parent->color = black;        % Case 1
        x->parent->parent->right->color = black;
        x->parent->parent->color = red;
        x = x->parent->parent; }
      else {
       if (x == x->parent->right) {      % Case 2
         x = x->parent;
         LeftRotate(T,x) }
       x->parent->color = black;         % Case 3
       x->parent->parent->color = red;
       RightRotate(T,x->parent->parent); }}
    else .... % same as above with right and left exchanged
  root->color = black;
```

For this program, we want to show that after an insertion of a node, a red-black tree remains a red-black tree. In this paper, we restrict ourselves to calculating the effects of program blocks which preserve the tree structure of the heap. This is not the case in general since pointer operations can temporarily break the tree structure, e.g. in the code for performing a rotation. The operations we handle are the following:

1. tests on the tree structure (like `x->parent == x->parent->parent->left`),
2. changing data of a node (as, e.g., recoloring of a node `x->color = red`),
3. left or right rotation (Figure 1 (b)),
4. moving a pointer up or down a tree structure (like `x = x->parent->parent`),
5. low-level insertion/deletion, i.e. the physical addition/removal of a node to/from a suitable place that is then followed by the re-balancing operations.

2 Preliminaries

In this paper, we work with the set $\mathfrak{D}$ of all boolean combinations of formulae of the form $x - y \diamond c$ or $x \diamond c$, for some $c \in \mathbb{Z}$ and $\diamond \in \{\leq, \geq\}$. We introduce the equality sign

as syntactic sugar, i.e. $x-y=c \iff x-y\leq c \wedge x-y\geq c$. Notice that negation can be eliminated from any formula of $\mathfrak{D}$, since $x-y\not\leq c \iff x-y\geq c+1$, and so on. Also, any constraint of the form $x-y\geq c$ can be equivalently written as $y-x\leq -c$. For a closed formula φ, we write $\models \varphi$ meaning that it is valid, i.e. equivalent to true.

A *ranked alphabet* Σ is a set of symbols together with a function $\# : \Sigma \to \mathbb{N}$. For $f \in \Sigma$, the value $\#(f)$ is said to be the *arity* of f. We denote by Σ_n the set of all symbols of arity n from Σ. Let λ denote the empty sequence. A *tree* t over an alphabet Σ is a partial mapping $t : \mathbb{N}^* \to \Sigma$ that satisfies the following conditions:

- $dom(t)$ is a finite prefix-closed subset of $\mathbb{N}^*$, and
- for each $p \in dom(t)$, if $\#(t(p)) = n > 0$ then $\{i \mid pi \in dom(t)\} = \{1,\ldots,n\}$.

A *subtree* of t starting at position $p \in dom(t)$ is a tree $t_{|p}$ defined as $t_{|p}(q) = t(pq)$ if $pq \in dom(t)$, and undefined otherwise. Given a set of positions $P \subseteq \mathbb{N}^*$, we define the *frontier* of P as the set $fr(P) = \{p \in P \mid \forall i \in \mathbb{N}\ pi \notin P\}$. For a tree t, we use $fr(t)$ as a shortcut for $fr(dom(t))$. We denote by $T(\Sigma)$ the set of all trees over the alphabet Σ.

Definition 1. *Given two trees $t : \mathbb{N}^* \to \Sigma$ and $t' : \mathbb{N}^* \to \Sigma'$, a function $h : dom(t) \to dom(t')$ is said to be a* tree mapping between t and t' *if the following hold:*

- $h(\lambda) = \lambda$, *and*
- *for any $p \in dom(t)$, if $\#(t(p)) = n > 0$ then there exists a prefix-closed set $Q \subseteq \mathbb{N}^*$ such that $pQ \subseteq dom(t')$ and $h(pi) \in fr(pQ)$ for all $1 \leq i \leq n$.*

A *size function* (or measure) associates to every tree $t \in T(\Sigma)$ an integer $|t| \in \mathbb{Z}$. Size functions are defined inductively on the structure of the tree. For each $f \in \Sigma$, if $\#(f) = 0$ then $|f|$ is a constant c_f, otherwise, for $\#(f) = n$, we have:

$$|f(t_1,\ldots,t_n)| = \begin{cases} b_1|t_1| + c_1 \text{ if } \models \delta_1(|t_1|,\ldots,|t_n|) \\ \quad\ldots \\ b_n|t_n| + c_n \text{ if } \models \delta_n(|t_1|,\ldots,|t_n|) \end{cases}$$

where $b_1,\ldots,b_n \in \{0,1\}$, $c_1,\ldots,c_n \in \mathbb{Z}$, and $\delta_1,\ldots,\delta_n \in \mathfrak{D}$, all depending on f. In order to have a consistent definition, it is required that $\delta_1,\ldots,\delta_n$ define a partition of $\mathbb{N}^n$, i.e. $\models \forall x_1 \ldots \forall x_n \bigvee_{1\leq i\leq n} \delta_i \wedge \bigwedge_{1\leq i<j\leq n} \neg(\delta_i \wedge \delta_j)$. [1] A *sized alphabet* $(\Sigma, |.|)$ is a ranked alphabet with an associated size function.

A *tree automaton with size constraints* (TASC) over a sized alphabet $(\Sigma, |.|)$ is a 3-tuple $A = (Q, \Delta, F)$ where Q is a finite set of states, $F \subseteq Q$ is a designated set of final states, and Δ is a set of transition rules of the form $f(q_1,\ldots,q_n) \xrightarrow{\varphi(|1|,\ldots,|n|)} q$, where $f \in \Sigma$, $\#(f) = n$, and $\varphi \in \mathfrak{D}$ is a formula with n free variables. For constant symbols $a \in \Sigma$, $\#(a) = 0$, the automaton has unconstrained rules of the form $a \to q$.

A *run* of A over a tree $t : \mathbb{N}^* \to \Sigma$ is a mapping $\pi : dom(t) \to Q$ such that, for each position $p \in dom(t)$, where $q = \pi(p)$, we have:

[1] For technical reasons related to the decidability of the emptiness problem for TASC, we do not allow arbitrary linear combinations of $|t_i|$ in the definition of $|f(t_1,\ldots,t_n)|$.

- if $\#(t(p)) = n > 0$ and $q_i = \pi(pi)$, $1 \leq i \leq n$, then Δ has a rule $t(p)(q_1,\ldots,q_n) \xrightarrow{\varphi(|1|,\ldots,|n|)} q$ and $\models \varphi(|t_{|p1}|,\ldots,|t_{|pn}|)$,
- otherwise, if $\#(t(p)) = 0$, then Δ has a rule $t(p) \to q$.

A run π is said to be *accepting*, if and only if $\pi(\lambda) \in F$. As usual, the *language* of A, denoted as $\mathcal{L}(A)$ is the set of all trees over which A has an accepting run.

As an example, let us consider a TASC recognising the set of all balanced red-black trees. Let $\Sigma = \{red, black, nil\}$ with $\#(red) = \#(black) = 2$ and $\#(nil) = 0$. First, we define the size function to be the maximal number of black nodes from the root to a leaf: $|nil| = 1$, $|red(t_1,t_2)| = max(|t_1|,|t_2|)$, and $|black(t_1,t_2)| = max(|t_1|,|t_2|)+1$. Let $A_{rb} = (\{q_b,q_r\},\Delta,\{q_b\})$ with $\Delta = \{nil \to q_b, black(q_{b/r},q_{b/r}) \xrightarrow{|1|=|2|} q_b, red(q_b,q_b) \xrightarrow{|1|=|2|} q_r\}$. By using $q_{x/y}$ within the left-hand side of a transition rule, we mean the set of rules in which either q_x or q_y take the place of $q_{x/y}$.

3 Closure Properties and Decidability of TASC

This section is devoted to the closure of the class of TASC under the operations of union, intersection and complement. The decidability of the emptiness problem is also proved.

3.1 Closure Properties

A TASC is said to be *deterministic* if, for every input tree, the automaton has at most one run. For every TASC A, we can effectively construct a deterministic TASC A_d such that $\mathcal{L}(A) = \mathcal{L}(A_d)$. Concretely, let $A = (Q,\Delta,F)$ and $\mathcal{G}_A$ be the set of all guards labelling the transitions from Δ and $\mathcal{G}_A^n = \{\varphi \in \mathcal{G}_A \mid \|FV(\varphi)\| = n\}$ where $n \in \mathbb{N}$ and $\|FV(\varphi)\|$ denotes the number of free variables in φ. Without loss of generality, we assume that any guard φ labelling a transition of A of the form $f(q_1,\ldots,q_n) \xrightarrow{\varphi} q$ has exactly n free variables.[2] Define $\mathcal{B}_A^n$ as the set of all conjunctions of formulae from $\mathcal{G}_A^n$ and their negations. Let $\mathcal{B}_A = \bigcup_{n\in\mathbb{N}} \mathcal{B}_A^n \cup \{\top\}$. With this notation, define $A_d = (Q_d,\Delta_d,F_d)$ where $Q_d = \mathcal{P}(Q) \times \mathcal{B}_A$, $F_d = \{\langle s,\varphi\rangle \in Q_d \mid s \cap F \neq \emptyset\}$, and:

$$f(\langle s_1,\varphi_1\rangle \ldots \langle s_n,\varphi_n\rangle) \xrightarrow{\varphi} \langle s,\varphi\rangle \in \Delta_d$$

$$\text{iff} \begin{cases} s \subseteq \{q \mid f(q_1,\ldots,q_n) \xrightarrow{\psi} q \in \Delta, q_i \in s_i\} \text{ and } s \neq \emptyset \\ \varphi = \bigwedge \{\psi \mid f(q_1,\ldots,q_n) \xrightarrow{\psi} q \in \Delta, q_i \in s_i, q \in s\} \wedge \\ \quad \bigwedge \{\neg\psi \mid f(q_1,\ldots,q_n) \xrightarrow{\psi} q \in \Delta, q_i \in s_i, q \in Q \setminus s\} \end{cases}$$

$$a \to \langle s,\top\rangle \in \Delta_d \text{ iff } s = \{q \mid a \to q \in \Delta\}$$

Notice that A_d has no states of the form $\langle s,\bot\rangle$ since they would necessarily be unreachable. The following theorem shows that non-deterministic and deterministic TASC recognize exactly the same languages (for a proof of the theorem see [10]).

[2] We can add conjuncts of the form $x_i = x_i$ for all missing variables.

Theorem 1. *A_d is deterministic and $\mathcal{L}(A_d) = \mathcal{L}(A)$.*

Determinisation is crucial to show closure of TASC under language complementation. However, given a deterministic TASC A, the construction of a TASC recognizing the language $T(\Sigma) \setminus \mathcal{L}(A)$ is fairly standard [6], using the fact that $\mathfrak{D}$ is closed under negation. One needs to first build the *complete* TASC, i.e. in which each input leads to one state, and then switch between accepting and non-accepting states. Fairly standard is also the union of TASC, i.e. given A_1 and A_2, one can build a TASC $A_\cup$ recognizing $\mathcal{L}(A_1) \cup \mathcal{L}(A_2)$ by simply merging their (supposedly disjoint) sets of states and transitions. The TASC $A_\cap$ recognizing intersection of languages, i.e. $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$, is the automaton whose set of states is the cartesian product of the sets of states of A_1 and A_2, and the transitions are of the form $f((q'_1,q''_1),\ldots,(q'_n,q''_n)) \xrightarrow[A_\cap]{\varphi' \wedge \varphi''} (q',q'')$, for $f(q'_1,\ldots,q'_n) \xrightarrow[A_1]{\varphi'} q'$ and $f(q''_1,\ldots,q''_n) \xrightarrow[A_2]{\varphi''} q''$. For more details, we refer the reader to the full version of the paper [10].

3.2 Emptiness

In this section, we give an effective method for deciding emptiness of a TASC. In fact, we address the slightly more general problem: given a TASC $A = (Q,\Delta,F)$ we construct, for each state $q \in Q$, an arithmetic formula $\phi_q(x)$ in one variable that precisely characterizes the sizes of the trees whose roots are labelled with q by A, i.e. $\models \phi_q(n)$ iff $\exists t\ |t| = n$ and $t \xrightarrow[A]{*} q$. As it will turn out, the ϕ_q formulae are expressible in Presburger arithmetic, therefore satisfiability is decidable [14]. This entails the decidability of the emptiness problem, which can be expressed as the satisfiability of the disjunction $\bigvee_{q \in F} \phi_q$.

In order to construct ϕ_q, we shall translate our TASC into an Alternating Pushdown System (APDS) [3] whose stack encodes the value of one integer counter. An APDS is a triple $S = (Q,\Gamma,\delta,F)$ where Q is the set of control locations, Γ is the stack alphabet, F is the set of final control locations, and δ is a mapping from $Q \times \Gamma$ into $\mathcal{P}(\mathcal{P}(Q \times \Gamma^*))$. Notice that APDS do not have an input alphabet since we are interested in the behaviors they generate, rather than the accepted languages. A run of the APDS is a tree $t : \mathbb{N}^* \to (Q \times \Gamma^*)$ satisfying the following property: for any $p \in dom(t)$, if $t(p) = \langle q, \gamma w\rangle$, then $\{t(pi) \mid 1 \leq i \leq \#(t(p))\} = \{\langle q_1, w_1 w\rangle, \ldots, \langle q_n, w_n w\rangle\}$ where $\{\langle q_1, w_1\rangle, \ldots, \langle q_n, w_n\rangle\} \in \delta(q,\gamma)$. The run is accepting if all control locations occurring on the frontier are final.

The idea behind the reduction is that any bottom-up run of a TASC on a given input tree can be mapped (in the sense of Definition 1) onto a top-down run of an APDS. The simulation invariant is that the size of a subtree from the run of the TASC is encoded by the corresponding stack in the run of the APDS. Next, we use the construction of [3] to calculate, for the given set of configurations σ, the set $\mathrm{pre}^*_q(\sigma)$ of configurations with control state q that have a successor set in σ, i.e. $c = \langle q, w\rangle \xrightarrow{*} C \subseteq \sigma$. It is shown in [3] that if σ is a regular language, then so is $\mathrm{pre}^*(\sigma)$, and the alternating finite automaton recognizing the latter can be constructed in time polynomial in the size of the APDS. Hence, the Parikh images of such $\mathrm{pre}^*_q(\sigma)$ sets are semilinear sets definable by Presburger formulae. In our case, $\sigma = \{\langle q, \varepsilon\rangle \mid q \in F\}$ is a finite set where ε is the (encoding

of the) empty stack. Using a unary encoding of the counter (as a stack), we obtain the needed formulae $\phi_q(x)$. For a detailed explanation, the reader is referred to [10].

Lemma 1. *For each TASC $A = (Q, \Delta, F)$ over a sized alphabet $(\Sigma, |.|)$ there exists an APDS $S_A = (Q_A, \Gamma, \delta, F_A)$ such that:*

1. *for any tree $t \in T(\Sigma)$ and any run $\pi : dom(t) \to Q$ of A on t, there exists an accepting run $\rho : \mathbb{N}^* \to (Q_A \times \mathbb{N})$ of S_A and a one-to-one tree mapping h between π and ρ such that:*
$$\forall p \in dom(t)\ \exists q \in Q_A\ .\ \rho(h(p)) = \langle q, |t_{|p}| \rangle \quad (1)$$
2. *for any accepting run $\rho : \mathbb{N}^* \to (Q_A \times \mathbb{N})$ of S_A there exists a tree $t \in T(\Sigma)$, a run $\pi : dom(t) \to Q$ of A on t and a one-to-one tree mapping h between π and ρ satisfying (1).*

Moreover, S_A can be effectively constructed from the description of A.

As a remark, the decidability of the emptiness problem for TASC can be also proved via a reduction to the class of *tree automata with one memory* [5] by encoding the size of a tree as a unary term, using essentially the same idea as in the reduction to APDS. The complexity of the emptiness problem can be furthermore analyzed using the double exponential bound of the emptiness problem for tree automata with one memory, and is considered as further work.

4 Semantics of Tree Updates

As explained in Section 1.1, there are three types of operations that commonly appear in procedures used for balancing binary trees after an insertion or deletion: (1) navigation in a tree, i.e. testing or changing the position a pointer variable is pointing to in the tree, (2) testing or changing certain data fields of the encountered tree nodes, such as the color of a node in a red-black tree, and (3) tree rotations. In addition, one has to consider the physical insertion or deletion to/from a suitable position in the tree as an input for the re-balancing.

It turns out that the TASC defined in Section 2 are not closed with respect to the effect of some of the above operations, in particular the ones that change the balance of subtrees (the difference between the size of the left and right subtree at a given position in the tree). Therefore, we now introduce a subclass of TASC called *restricted TASC* (rTASC) which we show to be closed with respect to all the needed operations on balanced trees. Moreover, rTASC are closed with respect to intersection and union, amenable to determinisation and minimization, though not closed with respect to complementation. The idea is to use rTASC to express loop invariants and pre- and post-conditions of programs as well as to perform the necessary reachability computations. TASC are then used in the associated language inclusion checks. Notice that, since rTASC are not closed under negation, inclusion of rTASC cannot be directly decided. Therefore we have to appeal to the more general result concerning the decidability of inclusion between TASC.

A *restricted alphabet* is a sized alphabet consisting only of nullary and binary symbols and a size function of the form $|f(t_1, t_2)| = \max(|t_1|, |t_2|) + a$ with $a \in \mathbb{Z}$ for binary

symbols. A *restricted* TASC is a TASC with a restricted alphabet and with binary rules only of the form $f(q_1,q_2) \xrightarrow{|1|-|2|=b} q$ with $b \in \mathbb{Z}$. Notice that any conjunction of guards of an rTASC and their negations reduces either to false, or to only one formula of the same form, i.e. $|1|-|2|=b$. Using this fact, one can show that the intersection of two rTASC is again an rTASC, and that applying the determinisation of Section 3.1 to an rTASC yields another rTASC. Moreover, the intersection of an rTASC with a classical tree automaton is again an rTASC.[3] Clearly, rTASC are not closed under complementation, as inequality guards are not allowed.

4.1 Representing Sets of Memory Configurations

Let us consider a finite set of *pointer variables* $\mathcal{V} = \{x,y,\ldots\}$ and a disjoint finite set of data values $\mathcal{D}$, e.g. $\mathcal{D} = \{red, black\}$. In the following, we let $\Sigma = \mathcal{P}(\mathcal{V} \cup \mathcal{D} \cup \{nil\})$ where *nil* indicates a null pointer value. The arity function is defined as follows: $\#(f) = 2$ if $nil \notin f$, and $\#(f) = 0$ otherwise. For a tree $t \in T(\Sigma)$ and a variable $x \in \mathcal{V}$, we say that a position $p \in dom(t)$ is *pointed to by* x if and only if $x \in t(p)$.

For the rest of this section, let $A = (Q, \Delta, F)$ be an rTASC over Σ. We say that A represents a *set of memory configurations* if and only if, for each $t \in L(A)$ and each $x \in \mathcal{V}$, there is at most one $p \in dom(t)$ such that $x \in t(p)$. This condition can be ensured by the construction of A: let $Q = Q \times \mathcal{P}(\mathcal{V})$ and Δ consist only of rules of the form $f(\langle q_1, v_1\rangle, \langle q_2, v_2\rangle) \xrightarrow{\varphi} \langle q, v\rangle$ where (1) $v = (f \cup v_1 \cup v_2) \cap \mathcal{V}$ and (2) $f \cap v_1 = f \cap v_2 = v_1 \cap v_2 = \emptyset$. Intuitively, a control state $\langle q, v\rangle$ "remembers" all variables encountered by condition (1), while condition (2) ensures that no variable is encountered twice.

4.2 Modelling Tree Rotations

Let $x \in \mathcal{V}$ be a fixed variable. We shall construct an rTASC $A' = (Q', \Delta', F')$ that describes the set of trees that are the result of the *left rotation* of a tree from $L(A)$ applied at the node pointed to by x. The case of the right tree rotation is very similar.[4] In the description, we will be referring to Figure 2 illustrating the problem.

Let $R_x = \{(r_1, r_2) \in \Delta^2 \mid x \in g \wedge r_1 : f(q_1,q_2) \xrightarrow{\varphi_3} q_3 \wedge r_2 : g(q_4,q_3) \xrightarrow{\varphi_5} q_5\}$ be the set of all the pairs of automata rules that can yield a rotation, and be modified because of it. Other rules may then have to be modified to reflect the change in one of their *left hand side states*, e.g. the change of q_5 to q_3' in the h-rule in Figure 2, or to reflect the *change in the balance* that may result from the rotation, i.e. a change in the *difference of the sizes* of the subtrees of some node. We discuss later what changes in the balance can appear after a rotation, and Lemma 2 proves that the set D of the possible changes in the balance in the described trees is finite. The automaton A' can thus be constructed from A as follows:

1. $Q' = Q \cup R_x \cup (R_x \times D) \cup (Q \times D)$ where we add new states for the rotated parts and to reflect the changes in the balance.

[3] A bottom-up tree automaton can be seen as a TASC in which all guards are true.

[4] In fact, it can be implemented by temporarily swapping the child nodes in the involved rules, doing a left rotation, and then swapping the child nodes again.

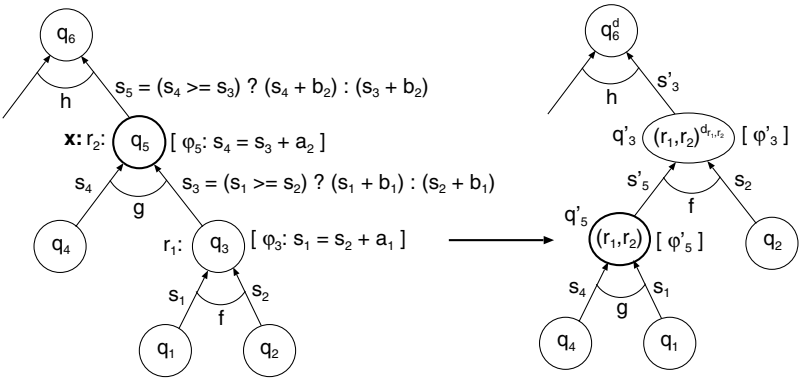

Fig. 2. Left rotation on an rTASC

2. $\Delta' = \Delta \cup \Delta_r \cup \beta(\Delta \cup \Delta_a)$ where:
   - Δ_r is the smallest set such that for all $(r_1, r_2) \in R_x$ where $r_1 : f(q_1, q_2) \xrightarrow{\varphi_3} q_3$ and $r_2 : g(q_4, q_3) \xrightarrow{\varphi_5} q_5$, contains the rules $g(q_4, q_1) \xrightarrow{\varphi'_5} q'_5$ and $f(q'_5, q_2) \xrightarrow{\varphi'_3} q'_3$ where $q'_5 = (r_1, r_2)$ and $q'_3 = (r_1, r_2)^{d_{r_1,r_2}}$. Here, we use $(r_1, r_2)^{d_{r_1,r_2}}$ as a shorthand for $\langle (r_1, r_2), d_{r_1,r_2} \rangle$. The value $d_{r_1,r_2} \in \mathbb{Z}$ represents the change in the balance caused by the rotation based on r_1, r_2. We describe the computation of φ'_3, φ'_5, and d_{r_1,r_2} below.
   - Δ_a is the set of rules that could be applied just above the position where a rotation takes place. For each $(r_1, r_2) \in R_x$, we take all rules from Δ that have q_5 within the left hand side and add them to Δ_a, with (r_1, r_2) substituted for q_5.
   - β (described in detail in Section 4.3) is the function that implements the necessary changes in the guards and input/output states (adding the d-component) of the rules due to the changes in the balance.
3. $F' = (F \times D) \cup F_r$. Here, F_r captures the case where q'_3 becomes accepting, i.e. the right child of the node previously labelled by q_3 becomes the root of the entire tree.

Suppose that φ_3 is $|t_1| = |t_2| + a_1$ and let us denote the sizes of the sub-trees read at q_1 and q_2 before the rotation by s_1 and s_2, respectively. Let the size function associated with f be $|f(t_1, t_2)| = \max(|t_1|, |t_2|) + b_1$, and let s_3 denote the size of the subtree labelled by q_3 before the rotation. Also, suppose that φ_5 is $|t_1| = |t_2| + a_2$ and let us denote the size of the sub-tree read at q_4 before the rotation as s_4. Finally, let the size function associated with g be $|g(t_1, t_2)| = \max(|t_1|, |t_2|) + b_2$, and let s_5 denote the size of the subtree labelled by q_5 before the rotation. We denote s'_5 and s'_3 the sizes obtained at q'_5 and q'_3 after the rotation.

The key observation that allows us to compute φ'_3, φ'_5, and d_{r_1,r_2} is that due to the chosen form of guards and sizes, we can always compute any two of the sizes s_1, s_2, s_4 from the remaining one. Indeed,

- for $a_1 \geq 0$, we have $s_3 = s_1 + b_1 = s_2 + a_1 + b_1 = s_4 - a_2$, whereas
- for $a_1 < 0$, we have $s_3 = s_2 + b_1 = s_1 - a_1 + b_1 = s_4 - a_2$.

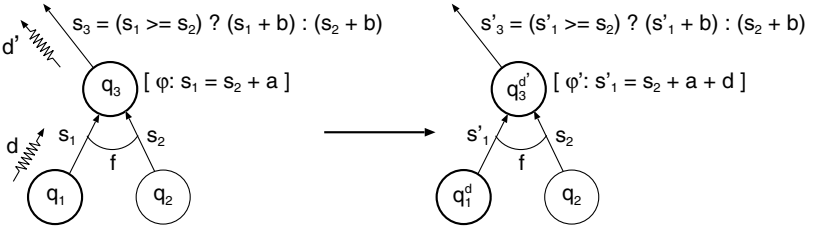

Fig. 3. Propagation of changes in the balance in an rTASC

Computing φ'_3, φ'_5, and d_{r_1,r_2} is then just a complex exercise in case splitting. Notice that all the cases can be distinguished statically according to the mutual relations of the constants a_1, b_1, a_2, and b_2. For example, in the case of φ'_5, we obtain the following (the other cases are explained in [10]):

1. For $a_1 \geq 0$, we have $s_4 = s_1 + b_1 + a_2$, and so φ'_5 relating a subtree of size s_4 and s_1 (cf. Figure 2) is $|t_1| = |t_2| + b_1 + a_2$.
2. For $a_1 < 0$, we have $s_4 = s_1 - a_1 + b_1 + a_2$, and so φ'_5 is $|t_1| = |t_2| - a_1 + b_1 + a_2$.

4.3 Propagating Changes in the Balance Through rTASC

As said, tree updates such as recoloring or rotations may introduce changes in the balance at certain points. These changes may affect the balance at all positions above the considered node. The role of the β function is to propagate a change in balance d upwards in the trees recognized by the rTASC. The way β changes a set of rules is illustrated in Figure 3. For every $d \in D$, every input rule $f(q_1,q_2) \xrightarrow{\varphi} q_3$ is changed to two rules $f(q_1^d,q_2) \xrightarrow{\varphi'} q_3^{d'}$ and $f(q_1,q_2^d) \xrightarrow{\varphi''} q_3^{d''}$ corresponding to the cases when the change in the balance originates from the left or the right. Since we consider just one rotation in every tree (at a given node pointed to by the pointer variable x), the change can never come from both sides. The new guards are $\varphi' : |t_1| = |t_2| + a + d$ and $\varphi'' : |t_1| = |t_2| + a - d$. Let us further analyse the changes in the balance propagated upwards after d comes from the bottom.

Suppose the change in balance is coming from the left as in Figure 3. We distinguish the cases of $a \geq 0$ and $a < 0$. (1) For $a \geq 0$, the original size at q_3 is $s_3 = s_1 + b$ where s_1 is the original size at q_1. After the change d happens at q_1, i.e. $s'_1 - s_1 = d$, we have the following subcases: (1.1) For $a + d \geq 0$, we have $s'_3 = s'_1 + b$, i.e. $d' = d$, and so we have the same change in the size at q_3 as at q_1. (1.2) For $a + d < 0$, we have $s'_3 = s_2 + b = s_1 - a + b$, and hence $d' = -a$. (2) For $a < 0$, $s_3 = s_2 + b$. In this case, (2.1) for $a + d \geq 0$, $s'_3 = s'_1 + b = s_1 + d + b = s_2 + a + d + b$, and so $d' = a + d$, and (2.2) for $a + d < 0$, $s'_3 = s_2 + b$, and thus $d' = 0$. The case of the change in the balance coming from the right is similar.

When a change d in the size happens at a child node, at its parent, the change is either eliminated, d' or d'' is 0, stays the same, d' or d'' equals d, becomes $-|a|$ (note that $a \geq 0$ for $d' = -a$, and $a < 0$, for $d'' = a$), or finally, becomes $-|a| + d$. We can now close our construction by showing that the set D of possible changes in the sizes is finite.

Lemma 2. *For an rTASC A over a set of variables $\mathcal{V}$ and a variable $x \in \mathcal{V}$, the set D of the possible changes in the balance generated by a left tree rotation at x is finite.*

Note that when we allow the use of two different constants b_f^1 and b_f^2 in the size function for binary nodes, the resulting class of automata will not be closed with respect to left or right rotations. It may happen that the changes in the balance could diverge, thus we would need an infinite number of compensating constants to be used for the different heights of the possible trees.

4.4 Other Operations on Sets of Trees Described by rTASC

It remains to show that in addition to tree rotations, rTASC are closed with respect to all the other needed operations on balanced trees listed in Section 1.1. Showing this fact is relatively simple, and so due to space limitations, we omit an exact description of this issue here and refer the reader to the full paper. In general, the remaining operations may be implemented by intersecting the given rTASC with a classical tree automaton encoding all the trees that fulfil a certain condition (such as `x->parent->left == x` or `x->parent->color == red`) and/or doing certain limited changes to the given rTASC. This includes changing the symbols read in certain rules (e.g., removing `x` from the symbol read in a certain rule and adding it to the symbol read in another rule when we move the pointer variable `x` in the tree) and adding, removing, and modifying certain simple rules to express the low-level insertion/deletion of nodes. Afterwards, we may possibly have to apply the function β from the previous section when the tree balance is changed.

To give an intuition on how an rTASC encoding a certain condition on pointers may look like, let us present the tree automaton describing the trees that fulfil the condition `x->parent->left == x`. We will have rules $f \to q_1$ and $g \to q_2$ for every $f, g \in \Sigma$ such that $\mathtt{x} \in g \setminus f$. We recall that $\Sigma = \mathcal{P}(\mathcal{V} \cup \mathcal{D} \cup \{nil\})$. Then, we have rules $f(q_1, q_1) \to q_1$, $g(q_1, q_1) \to q_2$, $f(q_2, q_1) \to q_3$, $f(q_3, q_1) \to q_3$, and $f(q_1, q_3) \to q_3$, with q_3 being the only accepting state. Here, the pointer referencing pattern gets simply captured in the rule $f(q_2, q_1) \to q_3$. An intersection with the described tree automaton may be used to implement the `if` statement testing the given condition. Intersections with similar tree automata may be used to isolate rules where certain changes of data, pointer locations, or insertion/deletion of a new node should happen.

5 Case Study: Red-Black Tree Insertion

To illustrate our methodology, we show how to prove an invariant for the main loop in procedure RB-Insert. (Note that all the steps are normally to be done fully automatically.) This invariant is needed to prove the correctness of the insertion procedure given in Section 1.1 that is, given a valid red-black tree as input to the procedure, the output is also a valid red-black tree. The invariant is the conjunction of the following facts:

1. x is pointing to a non-null node in the tree.
2. If a node is red, then (i) its left son is either black or pointed to by x, and (ii) its right son is either black or pointed to by x. This condition is needed as during the

re-balancing of the tree, a red node can temporarily become a son of another red node.
3. The root is either black or x is pointing to the root.
4. If x is not pointing to the the root and points to a node whose father is red, then x points to a red node.
5. Each maximal path from the root to a leaf contains the same number of black nodes. This is the last condition from the definition of red-black trees from Section 1.1.

For presentation purposes, if no guard is specified on a binary rule, we assume it to be $|1| = |2|$. Also, we denote singleton sets by their unique element, e.g. $\{red\}$ by red, and d_x stands for $\{d,x\}$, where $d \in \{red, black, nil\}$. Let $R = \{nil \to q_b, red(q_b,q_b) \to q_r, black(q_{b/r},q_{b/r}) \to q_b\}$. The loop invariant is given by the following rTASC A_1.

$$A_1 : F = \{q_{rx}, q_{bx}, q'_{bx}\},\ \Delta = R \cup \{black_x(q_{b/r},q_{b/r}) \to q_{bx}\ (1),\ black(q_{bx/rx},q_{b/r}) \to q'_{bx}(2),$$
$$black(q'_{bx/rx},q_{b/r}) \to q'_{bx},\quad black(q_{b/r},q'_{bx/rx}) \to q'_{bx}\ (3),\quad black(q_{b/r},q'_{bx/rx}) \to q'_{bx},$$
$$red_x(q_b,q_b) \to q_{rx},\quad red(q'_{bx},q_b) \to q'_{rx},\quad red(q_b,q'_{bx}) \to q'_{rx},$$
$$red(q_{rx},q_b) \to q'_{rx}\ (4),\quad red(q_b,q_{rx}) \to q'_{rx}\ (5)\}$$

Intuitively, q_b labels black nodes and q_r red nodes which do not have a node pointed to by x below them. q_{bx} and q_{rx} mean the same except that they label a node which is pointed to by x. Primed versions of q_{bx} and q_{rx} are used for nodes which have a subnode pointed to by x. In the following, this intuitive meaning of states will be changed by the program steps. We refer to the pseudo-code of Section 1.1.

We choose to illustrate Case 2 of the loop (the others are similar). If the loop entrance condition `x!= root && x->parent->color == red` is true, we obtain a new automaton A_2 given from A_1 by setting $F = \{q'_{bx}\}$ and by removing rules (1), (2), (3). After the condition `x->parent == x->parent->parent->left`, we get A_3 from A_2 by changing rule (4) to $red(q_{rx},q_b) \to q''_{rx}$, rule (5) to $red(q_b,q_{rx}) \to q''_{rx}$ and by adding $black(q''_{rx},q_{b/r}) \to q'_{bx}$. In Case 2, `x->parent->parent->right->color==red` is false, i.e. `x->parent->parent->right->color == black`. Applying this to A_3, we get:

$$A_4 : F = \{q'_{bx}\}, \Delta = R \cup \{black(q'_{bx/rx},q_{b/r}) \to q'_{bx},\quad black(q_{b/r},q'_{bx/rx}) \to q'_{bx},$$
$$black(q''_{rx},q_b) \to q'_{bx},\quad red_x(q_b,q_b) \to q_{rx}\ (8),\quad red(q'_{bx},q_b) \to q'_{rx},$$
$$red(q_b,q'_{bx}) \to q'_{rx},\quad red(q_b,q_{rx}) \to q''_{rx}\ (9),\quad red(q_{rx},q_b) \to q''_{rx}\ (7)\}$$

Now, q''_{rx} accepts the father of the node pointed by `x` and q'_{rx} its grandfather. After the condition `x == x->parent->right`, A_4 is changed into A_5 by removing rule (7). After `x = x->parent`, A_5 is changed into A_6 by changing rule (8) to $red(q_b,q_b) \to q_{rx}$ and rule (9) to $red_x(q_b,q_{rx}) \to q''_{rx}$. The operation `Left-Rotate(T,x)` introduces new states and transitions and we get the TASC A_7. Notice that no rebalancing is necessary.

$$A_7 : F = \{q'_{bx}\}, \Delta = R \cup \{black(q'_{bx/rx},q_{b/r}) \to q'_{bx},\quad black(q_{b/r},q'_{bx/rx}) \to q'_{bx},$$
$$black(q_{rot2},q_b) \to q'_{bx},\quad red_x(q_b,q_b) \to q_{rot1},\quad red(q'_{bx},q_b) \to q'_{rx},$$
$$red(q_b,q'_{bx}) \to q'_{rx},\quad red(q_{rot1},q_b) \to q_{rot2}\}$$

After `x->parent->color = black`, and the necessary propagation of the changes in the balance through the tree, we obtain:

$$A_8 : F = \{q'_{bx}\}, \Delta = R \cup \{black(q'_{bx/rx}, q_{b/r}) \xrightarrow{|1|=|2|+1} q'_{bx},\ red_x(q_b, q_b) \to q_{rot1},$$
$$black(q_{b/r}, q'_{bx/rx}) \xrightarrow{|1|+1=|2|} q'_{bx},\ red(q'_{bx}, q_b) \xrightarrow{|1|=|2|+1} q'_{rx},$$
$$black(q_{rot2}, q_b) \xrightarrow{|1|=|2|+1} q'_{bx}, red(q_b, q'_{bx}) \xrightarrow{|1|+1=|2|} q'_{rx},\ black(q_{rot1}, q_b) \to q_{rot2}\}$$

After `x->parent->parent->color = red`, we obtain:

$$A_9 : F = \{q'_{bx}\}, \Delta = R \cup \{black(q'_{bx/rx}, q_{b/r}) \to q'_{bx},\ red_x(q_b, q_b) \to q_{rot1}, black(q_{b/r}, q'_{bx/rx}) \to q'_{bx},$$
$$red(q'_{bx}, q_b) \to q'_{rx}, red(q_{rot2}, q_b) \xrightarrow{|1|=|2|+1} q'_{bx}, \qquad red(q_b, q'_{bx}) \to q'_{rx}, black(q_{rot1}, q_b) \to q_{rot2}\ \}$$

Finally, after `Right-Rotate(T,x->parent->parent)`, we get:

$$A_{10} : F = \{q'_{bx}\}, \Delta = R \cup \{black(q'_{bx/rx}, q_{b/r}) \to q'_{bx}, \quad black(q_{b/r}, q'_{bx/rx}) \to q'_{bx}$$
$$black(q_{b/r}, q_{rot4}) \to q'_{bx}, \quad black(q_{rot4}, q_{b/r}) \to q'_{bx}, \quad black(q_{rot1}, q_{rot3}) \to q_{rot4},$$
$$red_x(q_b, q_b) \to q_{rot1}, \quad red(q'_{bx}, q_b) \to q'_{rx}, \quad red(q_b, q'_{bx}) \to q'_{rx},$$
$$red(q_{rot4}, q_b) \to q'_{rx}, \quad red(q_b, q_b) \to q_{rot3}, \quad red(q_b, q_{rot4}) \to q'_{rx}\}$$

Now, it can be easily checked that $\mathcal{L}(A_{10}) \subseteq \mathcal{L}(A_1)$. Case 3 is then very similar to Case 2 and Case 1 is presented in [10].

6 Conclusions

We have presented a method for semi-algorithmic verification of programs that manipulate balanced trees. The approach is based on specifying program pre-conditions, post-conditions, and invariants as sets of trees recognized by a novel class of extended tree automata called TASC. TASC come with interesting closure properties and a decidable emptiness problem. Moreover, the semantics of tree-updating programs can be effectively represented as modifications on the internal structures of TASC. The framework has been validated on a case study consisting of the node insertion procedure in a red-black tree. Precisely, we verify that given a balanced red-black tree on the input to the insertion procedure, the output is again a balanced red-black tree.

In the future, we plan to implement the method to be able to perform more case studies. An interesting subject for further research is then extending the method to a fully automatic one. For this, a suitable acceleration method for the reachability computation on TASC is needed. Also, it is interesting to try to generalize the method to handle even the internals of low-level manipulations that temporarily break the tree shape of the considered structures (e.g., by lifting the technique to work over tree automata extended with routing expressions describing additional pointers over the tree backbone).

Acknowledgement. We would like to thank Eugene Asarin, Ahmed Bouajjani, Yassine Lakhnech, and Tayssir Touili for their valuable comments.

References

1. R. Alur and P. Madhusudan. Visibly Pushdown Languages. In *Proceedings of STOC'04*. ACM Press, 2004.
2. P. Baldan, A. Corradini, J. Esparza, T. Heindel, B. König, and V. Kozioura. Verifying Red-Black Trees. In *Proc. of COSMICAH'05*, 2005.
3. A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *Proceedings of CONCUR '97*, volume 1243 of *LNCS*. Springer, 1997.
4. C. Calcagno, P. Gardner, and U. Zarfaty. Context Logic and Tree Update. In *Proceedings of POPL'05*. ACM Press, 2005.
5. H. Comon and V. Cortier. Tree Automata with One Memory, Set Constraints and Cryptographic Protocols. *Theoretical Computer Science*, 331, 2005.
6. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. Release October 1, 2002.
7. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
8. S. Dal Zilio and D. Lugiez. Multitrees Automata, Presburger's Constraints and Tree Logics. Technical Report 08-2002, LIF, 2002.
9. D. Geidmanis. Unsolvability of the Emptiness Problem for Alternating 1-way Multi-head and Multi-tape Finite Automata over Single-letter Alphabet. In *Computers and Artificial Intelligence*, volume 10, 1991.
10. P. Habermehl, R. Iosif, and T. Vojnar. Automata-based Verification of Programs with Tree Updates. Technical Report TR-2005-16, Verimag, 2005.
11. A. Moeller and M. Schwartzbach. The Pointer Assertion Logic Engine. In *Proceeedings of PLDI'01*. ACM Press, 2001.
12. S. Parduhn. Algorithm Animation Using Shape Analysis with Special Regard to Binary Trees. Technical report, Universität des Saarlandes, 2005.
13. H. Petersen. Alternation in Simple Devices. In *Proceedings of ICALP'95*, volume 944 of *LNCS*. Springer, 1995.
14. M. Presburger. Über die Vollstandigkeit eines Gewissen Systems der Arithmetik. *Comptes Rendus du I Congrés des Pays Slaves*, Warsaw, 1929.
15. M.O. Rabin. Decidability of Second Order Theories and Automata on Infinite Trees. *Transactions of American Mathematical Society*, 141, 1969.
16. R. Rugina. Quantitative Shape Analysis. In *Proceedings of SAS'04*, volume 3148 of *LNCS*. Springer, 2004.
17. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *TOPLAS*, 24(3), 2002.
18. H. Seidl, T. Schwentick, A. Muscholl, and P. Habermehl. Counting in Trees for Free. In *Proceedings of ICALP'04*, volume 3142 of *LNCS*. Springer, 2004.